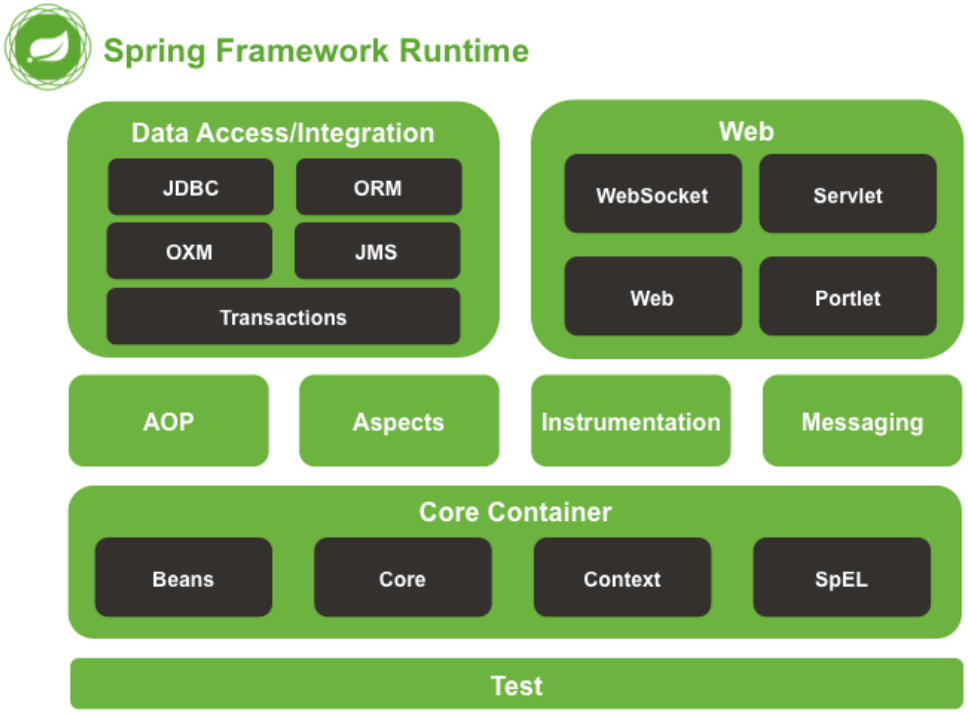
SPRING FRAMEWORKS



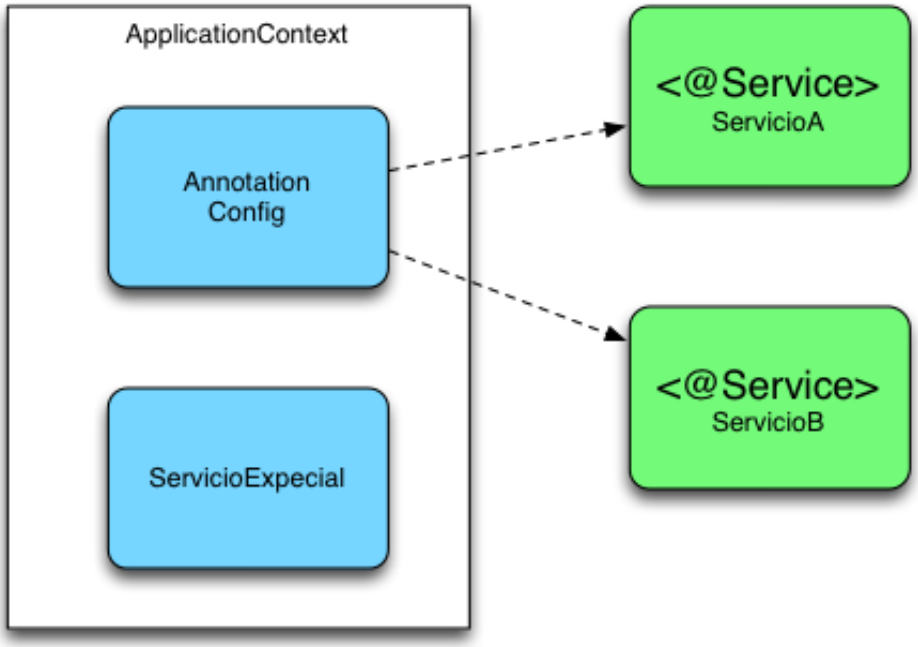
Capitulo 3
Spring Core Anotaciones

Contenido

1 INTRODUCCIÓN

1.1 Contexto

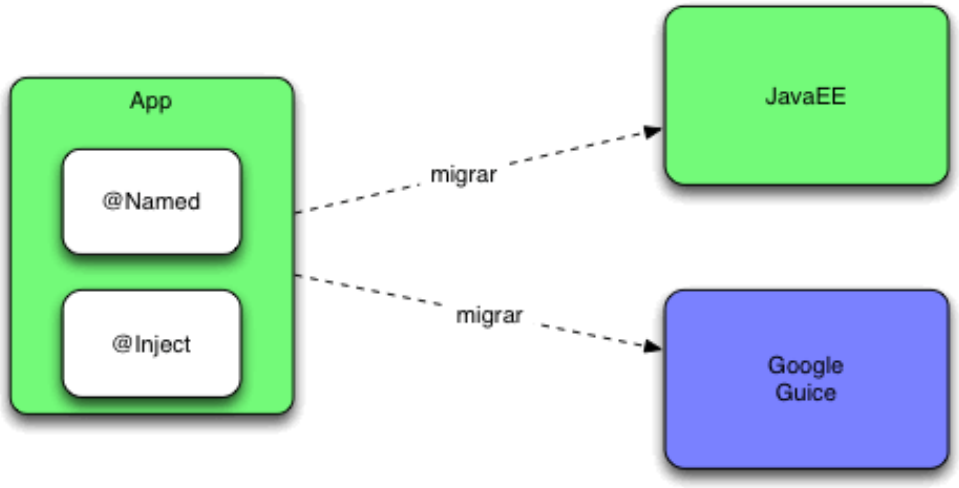
Usar anotaciones dentro de las clases permite que la configuración y la implementación estén en un único sitio.



Spring soporta que toda la configuración esté en un fichero XML, que se haga todo en forma de anotaciones, o que se mezclen ambos formatos.

Lo importante es recordar que primero se aplican las anotaciones y luego la configuración proveniente de los ficheros, pudiendo esta última sobreescribir lo establecido por las anotaciones.

La competencia entre los standares de Java EE y el framework Spring es cada vez más dura ya que las similitudes entre ambos son muchas. Elegir uno u otro depende de muchas cosas.



En estos momentos existen muchos proyectos de Spring y algunos de ellos están valorando pasarse a Java EE en futuras evoluciones. Por ejemplo proyectos con JSF ya que la integración en Java EE es más sencilla al pertenecer al propio estandar. Si nos encontramos en situaciones como estas o similares. Podemos programar la aplicación de Spring para que use anotaciones que están dentro de los standares de de Java EE y nos facilite las migraciones.

1.2 Configuración

Para que Spring tenga en cuenta las anotaciones hay que añadir las siguientes etiquetas den el archivo applicationContext.xml:

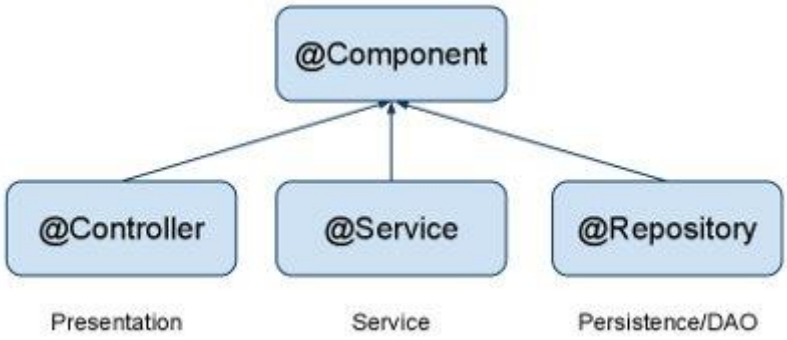
- **annotation-config**: Para activar el uso de anotaciones.
- **component**-scan: Para indicar el paquete donde Spring buscara las clases que se encuentren anotadas.

A continuación tenemos un ejemplo:

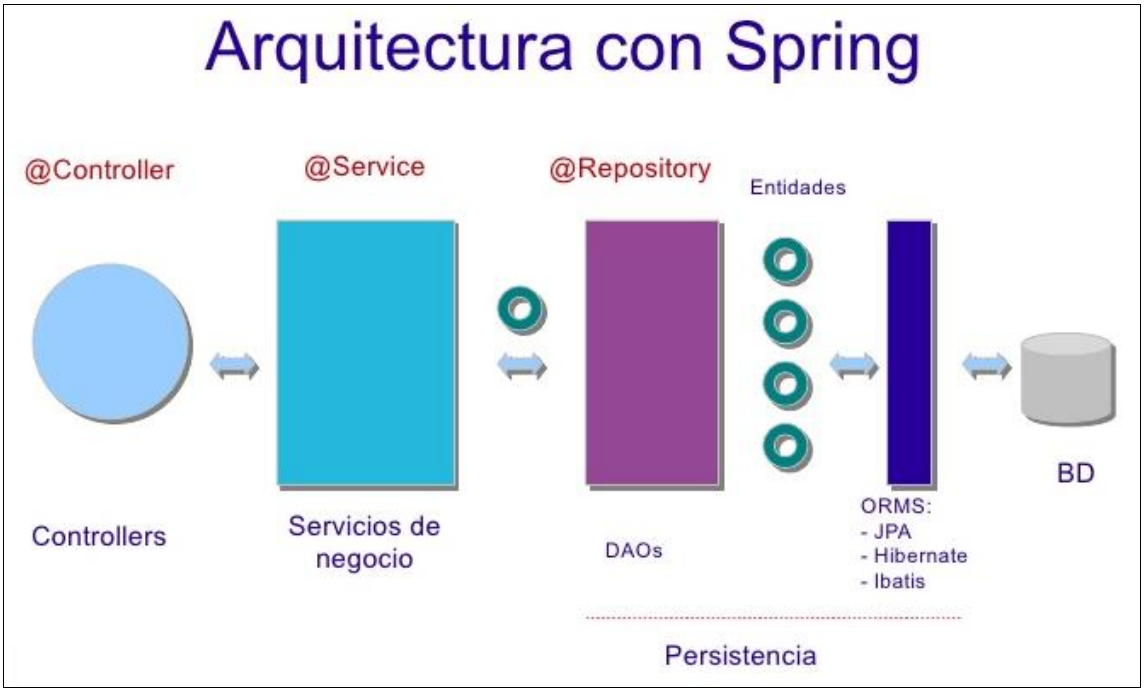
```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.1.xsd">

    <context:annotation-config />
    <context:component-scan base-package="pe.egcc.app.spring.beans"/>

    . . .

</beans>
```

2 COMPONENTES

Las anotaciones pueden utilizarse para que Spring detecte las clases importantes dentro del propio código fuente de Java evitando tener que declararlas en ficheros XML.



A continuación tenemos la arquitectura de como se deben utilizar las anotaciones:



2.1 @Component

Esta anotación sirve para añadir un estereotipo de forma genérica a una clase. Un "estereotipo" es una manera de clasificar las clases a un alto nivel. Es información normalmente de tipo semántica, pero puede utilizarse para caracterizar una clase al punto de establecer como debe comportarse cuando se produzca una excepción, por ejemplo.

```
import org.springframework.stereotype.Component;

@Component
public class Motor {
    . . .
    . . .
}
```

La anotación @Component es genérica y no se espera que se utilice realmente, lo esperado es que usen algunas de las otras anotaciones derivadas de ellas como @Repository, @Service y @Controller, para la capa de persistencia, servicios y vista respectivamente.

2.2 @Named

Esta anotación es similar a la anterior @Component. La ventaja está en que pertenece a la especificación estándar de Java en vez de pertenecer a Spring.

```
import javax.inject.Named;

@Named
public class Motor {
    . . .
    . . .
}
```

No obstante, para utilizar esta notación es necesario añadir la dependencia correspondiente en el fichero pom.xml:

```
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

3 DEPENDENCIAS

3.1 @Required

Esta anotación permite indicar que una dependencia debe ser resuelta obligatoriamente en tiempo de configuración. Es decir, que una determinada propiedad de un bean debe tener un valor distinto de nulo al acabar de instanciarla.

Consideremos el ejemplo de un coche que necesita de su motor:

```
import org.springframework.beans.factory.annotation.Required;

public class Coche {

    private Motor motor;

    @Required
    public void setMotor(Motor motor) {
        this.motor = motor;
    }

    . . .
}
```

Si en el applicationContext.xml definimos el bean de esta forma:

```
    <bean id="coche" class="com.empresa.Coche"/>
```

Al ejecutar la aplicación se elevará una excepción de tipo BeanCreationException porque la dependencia que tiene el coche con el motor no está resuelta.

Para evitarlo es necesario activar algún tipo de autodescubrimiento o inyectarla explícitamente:

```
<bean id="motor" class="com.empresa.Motor">

<bean id="coche" class="com.empresa.Coche">
    <property name="motor" ref="motor"/>
</bean>
```

La anotación @Required se aplica comúnmente sobre los DAOs a los que accede un servicio.

3.2 @Autowired

Esta anotación permite el autodescubrimiento e inyección automática de dependencias. Resuelve el problema del apartado anterior, haciendo innecesario declarar de forma explícita las dependencias en el fichero XML de configuración, y además tiene la ventaja de que se puede aplicar sobre muchos de los elementos de una clase.

El uso más común es sobre un setter, de forma que Spring automáticamente busca el bean que mejor se dapta al tipo del parámetro:

```
@Autowired
public void setMotor(Motor motor) {
    this.motor = motor;
}
```

Pero también se puede utilizar sobre las propiedades directamente, sobre el constructor, o sobre un método, con cualquier nombre y con cualquier cantidad y tipo de parámetros:

```
@Autowired
private Motor motor;

@Autowired
public Coche(Motor motor) {
    this.motor = motor;
}

@Autowired
public void montaje(Motor motor, Volante volante)   {
    this.motor = motor;
    this.volante = volante;
}
```

Incluso se puede utilizar para obtener todos los beans de un mismo tipo declarados en la configuración y que se almacenen en algún tipo de colección:

```
@Autowired
private Pieza[] piezas;

@Autowired
private List<Pieza> piezas;

@Autowired
private Map<String, Pieza> piezas;
```

Si la dependencia no se puede resolver se eleva una excepción. No obstante, este comportamiento se puede modificar utilizando el parámetro required:

```
@Autowired(required=false)
private Copiloto copiloto;
```

3.3 @Inject

Esta anotación tiene el mismo comportamiento que la anterior @Autowired, aunque carece del parámetro required. La ventaja está en que pertenece a la especificación estándar de Java en vez de pertenecer a Spring.

```
import javax.inject.Inject;

public class Coche {

    @ Inject
    private Motor motor;

    ...
    ...

}
```

No obstante, para utilizar esta notación es necesario añadir la dependencia correspondiente en el fichero pom.xml:

```
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

3.4 @Resource

Esta anotación se utiliza para eliminar ambigüedades a la hora de inyectar dependencias automáticamente, sobre todo si se aplica la anotación **@Autowired** en propiedades que tienen como tipo una interface, ya que clases distintas pueden implementar una misma interface.

```
@Autowired
@Resource(name="volante")
private Pieza volante;

@Autowired
@Resource(name="retrovisor")
private Pieza retrovisor;
```

Utiliza los identificadores de los beans para resolver las dependencias. Pertenece a la especificación estándar de Java, en vez de a Spring, por lo que requiere añadir la misma dependencia de Maven indicada para la anotación @Inject.

```
<bean id="volante" class="com.empresa.Pieza"/>
<bean id="retrovisor" class="com.empresa.Pieza"/>
```

Si se omite el valor del parámetro name se intenta resolver la dependencia buscando un bean que tenga el mismo nombre de la propiedad a la que se ha aplicado la anotación. Y si no se encuentra un bean que case se intenta resolver por tipo.

3.5 @Qualifier

Esta anotación tiene un comportamiento similar a la anterior @Resource, pero utiliza los roles de los beans en vez de sus identificadores para resolver las dependencias.

Su uso más común es aplicarla usando el valor de alguna característica semántica de los beans definidos en la configuración. Así dicho asusta un poco, pero la "característica semántica" es simplemente el valor de la etiqueta qualifier que tiene un bean dentro del fichero XML, y que se usa para indicar el "rol" que tiene un bean dentro de la aplicación.

Supongamos la siguiente configuración de ejemplo donde los beans representan piezas de un coche que se clasifican en función de su posición:

```
<bean id="faros" class="com.empresa.Pieza">
    <qualifier value="frontal"/>
</bean>

<bean id="parachoque" class="com.empresa.Pieza">
    <qualifier value="frontal"/>
</bean>

<bean id="maletero" class="com.empresa.Pieza">
    <qualifier value="posterior"/>
</bean>
```

Utilizando la anotación @**Qualifier** se puede eliminar la ambiguedad a la hora de resolver las dependencias de forma automática, incluso cuando se aplican a colecciones:

```
@Autowired
@Qualifier("posterior")
private Pieza maletero;

@Autowired
@Qualifier("frontal")
private List<Pieza> morro;
```

Las anotaciones son un mecanismo muy potente proporcionado por Java, a la vez que extensible, lo que hace incluso posible utilizar anotaciones @Qualifier propias en vez de ceñirse a las existentes por defecto.

4 AUTODETECCIÓN

Hasta ahora todos los ejemplos vistos han implicado siempre tener que declarar los beans en el fichero applicationContext.xml. Sin embargo, gracias a las anotaciones de tipo componente este tipo de declaraciones se pueden omitir y dejar que Spring detecte automáticamente los beans.

Supongamos las siguientes clases de ejemplo de una biblioteca:

```
public class Libro {

    . . .

}

public interface LibroDao {

    Libro getLibro();

}

@Repository
public class LibroDaoImpl implements LibroDao {

    public Libro getLibro() {
        return new Libro();
    }

}

@Service
public class Biblioteca {

    private LibroDao libroDao;

    @Autowired
    public void setLibroDao(LibroDao libroDao) {
        this.libroDao = libroDao;
    }
```

```
    public Libro getLibro() {
        return libroDao.getLibro();
    }

}
```

En este ejemplo hay dos componentes que nos interesa dejar que sea Spring quien los gestione automáticamente. Por una parte el repositorio de libros, y por otra parte el servicio de biblioteca. Para conseguir este propósito se define el fichero applicationContext.xml de la siguiente forma:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.1.xsd">

    <context:annotation-config />

    <context:component-scan base-package="com.empresa"/>

</beans>
```

Es decir, se utiliza la etiqueta **context:component-scan**, con el nombre del paquete donde se encuentran las clases, para dejar que Spring se encargue de escanearlo por sí solo, y que extraiga la definición de los beans evitando tener que hacerlo manualmente.

Para comprobar su correcto funcionamiento se pueden ejecutar las siguientes líneas de código donde se observa como un bean llamado "biblioteca" es automáticamente creado por String, a la par que todas sus dependencias han sido correctamente inyectadas:

```
Biblioteca biblioteca = context.getBean("biblioteca", Biblioteca.class);
biblioteca.getLibro();
```

Cuando se usa la etiqueta context:component-scan para escanear componentes no es necesario incluir también la etiqueta context:annotation-config para detectar anotaciones, esta última está incluida por defecto en el comportamiento de la primera.

Por último, comentar que si se quieren excluir algunas clases de ser automáticamente detectadas por Spring se pueden establecer filtros en la misma etiqueta usando nombres de clases, interfaces y anotaciones, e incluso expresiones regulares, expresiones de tipo de AspectJ, y filtros personalizados.

5 CONFIGURACIÓN

Los anteriores apartados se han mostrado como usar anotaciones para embeber algunos detalles de configuración en el propio código Java. En este apartado se muestra como especificar toda la configuración completa dentro de una única clase Java equivalente a un fichero de configuración XML.

5.1 @Configuration y @Bean

La anotación **@Configuration** indica que la clase sobre la que se encuentra aplicada debe ser usada como parte de la configuración de Spring. La anotación **@Bean** indica que el elemento sobre el que se encuentra aplicada define un bean.

Consideremos un ejemplo sencillo de una aplicación que implementa un único servicio de reparación de coches:

```
public class Coche {

    . . .

}

public interface Garage {

    void repara(Coche coche);

}

public class GarageImpl implements Garage {

    public void repara(Coche coche) {

        . . .

    }

}
```

```
@Configuration
public class AppConfig {

    @Bean
    public Garage garageBean() {
        return new GarageImpl();
    }

}
```

Lo interesante de esta definición de clases es que no requiere crear un fichero applicationContext.xml para que Spring funcione. Lo único que hay que hacer es cambiar el método main que estábamos utilizando desde el principio, para indicar que se quiere utilizar una clase de configuración en vez de un fichero:

```
    AnnotationConfigApplicationContext context =
        new AnnotationConfigApplicationContext(AppConfig.class);
    Garage garage = context.getBean(Garage.class);
```

La clase AppConfig es equivalente al siguiente fichero de configuración:

```
    <bean id="garageBean" class="com.empresa.GarageImpl"/>
```

Los atributos que normalmente se utilizan sobre un bean en un fichero de configuración pueden añadirse dentro de la anotación @Bean:

```
    @Bean(name="Chapuzas S.A.", initMethod="init", destroyMethod="exit")
    public Garage garageBean() {

        . . .

    }
```

La propia clase de configuración queda registrada como un bean, así como cualquier otra clase marcada con el estereotipo de componente, y sobre ellas las anotaciones de inyección de dependencias automáticas se aplican de la forma habitual.

Además, se pueden añadir tantas clases al contexto como se quiera. Ya sea inyectándolas como parámetros en el constructor, o registrándolas más tarde por código en tiempo de ejecución:

```
context.register(Configuracion.class);
context.register(Componente.class);
context.refresh();
```

También se le puede dejar a Spring la tarea de escanear un paquete de clases, de igual forma que ocurre cuando se específica la etiqueta context:component-scan en los ficheros XML:

```
context.scan("com.empresa");
context.refresh();
```

5.2 @Import

Esta anotación permite que una clase de configuración incluya a otra. Esto para evitar tener que registrarlas una a una. Basta con hacer una principal que incluya a todas las demás:

```
@Configuration
@Import(AppConfigAuxiliar.class)
public class AppConfig {

    ...

}
```

5.3 @ImportResource

Esta anotación es similar a la anterior, pero permite especificar el nombre de un fichero (recurso) en vez de una clase:

```
@Configuration
@ImportResource("classpath:beans.xml")
public class AppConfig {

    ...

}
```

5.4 @Scope

Esta anotación permite especificar el tipo de un bean. Es decir, el alcance de su definición, de igual forma que la etiqueta scope en los ficheros XML:

```
@Bean
@Scope("prototype")
public Coche cocheBean() {
    . . .
}
```